Page Replacement Policies

Page Replacement — Naive

- **step 1: save/clear victim page**:
  - drop page if fetched from disk and clean
  - *dirty*: write back modifications if from disk and dirty (unless `MAP_COPY`)
  - *non-dirty*: write page file/swap partition otherwise (e.g., stack, heap memory)
- **step 2: unmap page from old AS**: invalidate PTE, flush cache
- **step 3: prepare new page**: null page or load new contents
- **step 4: map page frame into new AS**: invalidate PTE, flush cache

Page Replacement — Buffering

- **Problem**: naive page replacement encompasses two I/O transfers

  → both operations block page fault from completing
- **Goal**: reduce I/O from critical page fault path to speed up page faults
- **Idea**: keep pool of free page frames (*pre-cleaning*):
  - *on page fault*: use page frame from free pool
  - *cleaning*: daemon cleans, reclaims and scrubs pages for free pool in background

  → smooths out I/O, speeds up paging significantly
- **Remaining problem**: which pages to select as victims?
  - *goal*: identify page that has left working set of its processes, add to free pool
  - *success metric*: low overall page fault rate

Page Replacement — FIFO

- **Idea**: evict oldest fetched page in system
- **Belady's Anomaly**: using FIFO, for every number n of page frames you can construct a reference string that performs worse with $n + 1$ frames

  → with FIFO it is possible to get more page faults with more page frames!

Page Replacement — oracle

= optimal replacement strategy: replace page with next reference furthest in future
- **Problem**: future unpredictable
- **However**: good metric to check how well other algorithms perform

Page Replacement — LRU

- **Goal**: approximate oracle page replacement
- **Idea**: past often predicts future well
- **Assumption**: page used furthest in past is used furthest in future
- **Cycle counter implementation**:
  - have MMU write CPU's time stamp counter to PTE on every access
  - *page fault*: scan all PTEs to find oldest counter value
  - *advantage*: cheap at access if done in HW
  - *disadvantage*: memory traffic for scanning
- **Stack implementation**:
  - keep doubly linked list of all page frames
  - move each referenced page to tail of list
  - *advantage*: can find replacement victim in $O(1)$
  - *disadvantage*: need to change 6 pointers at every access
- ⇝ **No silver bullet**:
  - *observation*: predicting future based on past is not precise
  - *conclusion*: relax requirements — maybe perfect LRU isn't needed? ⇒ approximate LRU

LRU Approximation — Clock page replacement

- aka *second chance page replacement*
- **Precondition**: MMU sets reference bit in PTE
  - supported natively by most hardware
  - can easily emulate in systems with software managed TLB (e.g., MIPS)
- **Store**: keep all pages in circular FIFO list
- **Searching** for victim: scan pages in FIFO's order
  - if reference bit = 0 → use page as victim and advance
  - if reference bit = 1 → set to 0, continue scanning
- **Problem**: large memory → most pages referenced before scanned
  - *solution*: use 2 arms, leading arm clears reference bit, trailing arm selects victim

Replacement Strategies — other

- **Random Eviction**: pick random victim
  - dirt simple
  - not overly horrible in reality
- **Larger counter**: use n-bit reference counter instead of reference bit
  - *least frequently used*: rarely used page not in a working set → replace page with smallest count
  - *most frequently used*: page with smallest count probably just brought in → replace page with largest count
  - neither LFU nor MDU are common (no such hardware, not that great)

Summary

- victim page frame needs to be selected by OS when handling page faults
  - evicting page frame after page fault happens = not a good idea
  - page buffering keeps eviction out of critical path
- different victim selection policies exist
  - FIFO → Belady's Anomaly
  - Oracle → cannot predict the future
  - Random → unpredictable, never great but rarely very bad
  - LRU → hard to implement efficiently